Lab3 Report

线程管理

对于 `cap_group.c` 中的函数，均可以通过注释的提示，调用对应的工具函数来完成。

对于 `thread.c` 中的 `create_root_thread`，使用 `create_pmo` 来分配物理页，使用对应的内存操作函数 `memcpy` 和 `memset` 来完成程序镜像的写入内存，只需要注意地址的计算。

对于 `context.c` 的初始化进程上下文函数，需要查找对应数据结构 `thread_ctx` 的实现来完成。

思考题 1
内核从完成必要的初始化到第一次切换用户态程序的过程是怎么样的，描述一下调用关系。

> - 必要的初始化主要包括完成内存管理、中断初始化和调度器初始化，随后会调用 `create_root_thread()` 来创建第一个进程。
> - 第一个进程为进程管理器 `procmgr`， `create_root_thread()` 内部首先、`create_root_cap_group` 来创建根进程的权利组，并为进程分配物理页，通过虚拟页映射将程序镜像写入对应的物理页。随后调用 `thread_init()` 来创建第一个线程，初始化线程上下文，并将其加入进程的权利组中，该线程会加载镜像中的 elf 信息（就是 procmgr 程序）。
> - 调用 `sched()` 和 `eret_to_thread()` 由于此时只有一个线程，刚刚创建的线程会被选中，这就切换到了用户态程序。

异常管理

`irq_entry.S` 中异常向量表根据提供的表格填入即可。

系统调用

- `exception_enter`：进入异常时需要储存所有的寄存器。
- `exception_exit`：退出异常时需要恢复储存的所有寄存器。
- `switch_to_cpu_stack`：切换为 CPU 栈则使用预先的宏定义来计算偏移量，将 `sp` 切换为 CPU 栈的地址。

思考题 2
尝试描述 `printf` 如何调用到 `chcore_stdout_write` 函数

> - `printf` 函数调用了 `vfprintf`，其中文件描述符参数为 `stdout`。这说明在 `vfprintf` 中将使用 `stdout` 的某些操作函数

> - `vfprintf` 函数中调用了 `stdout` 的 `write` 函数
> - 在 `user/chcore-libc/libchcore/porting/overrides/src/chcore-port/stdio.c` 中可以看到 `stdout` 的 `write` 操作被定义为 `__stdout_write` 。
> - `__stdout_write` 又调用了 `__stdio_write` 函数。
> - `__stdio_write` 则以 `stdout` 为文件描述符，调用了系统指令 `SYS_writev` ，对应于函数 `chcore_writev` 。
> - `chcore_write` 则通过 `fd_dic[fd]->fd_op->write(fd, buf, count)` 语句调用到函数 `chcore_stdout_write` 。

需要填写的内容位于 `user/chcore-libc/libchcore/porting/overrides/src/chcore-port/stdio.c` ， `put` 函数将调用 `chcore_syscall2` ，参数可以通过查找对应的头文件获得。

用户态程序编写

写一个简单的 C 程序

```c
#include <stdio.h>

int main() {
    printf("Hello ChCore!\n")
    return 0;
}
```

并调用 `build/chcore-libc/bin/musl-gcc` 进行编译，将编译后的 `hello_world.bin` 放入 `ramdisk/` 即可，内核在编译时会将程序自动写入磁盘。